Brief Announcement: A Persistent Lock-Free Queue for Non-Volatile Memory*

Michal Friedman[1], Maurice Herlihy[2], Virendra Marathe[3], and Erez Petrank[4]

1 **Technion, Haifa, Israel**
michal.f@cs.technion.ac.il
2 **Brown University, Providence, RI, USA**
mph@cs.brown.edu
3 **Oracle Labs, Redwood Shores, CA, USA**
virendra.marathe@oracle.com
4 **Technion, Haifa, Israel**
erez@cs.technion.ac.il

Abstract

Non-volatile memory is expected to coexist with (or even displace) volatile DRAM for main memory in upcoming architectures. As a result, there is increasing interest in the problem of designing and specifying *durable* data structures that can recover from system crashes. Data-structures may be designed to satisfy stricter or weaker durability guarantees to provide a balance between the strength of the provided guarantees and performance overhead. This paper proposes three novel implementations of a concurrent lock-free queue. These implementations illustrate the algorithmic challenges in building persistent lock-free data structures with different levels of durability guarantees. We believe that by presenting these challenges, along with the proposed algorithmic designs, and the possible levels of durability guarantees, we can shed light on avenues for building a wide variety of durable data structures. We implemented the various designs and evaluate their performance overhead compared to a simple queue design for standard (volatile) memory.

1998 ACM Subject Classification E.1 Data Structures, D.4.2 Storage Management, D.1.3 Concurrent Programming

Keywords and phrases Non-volatile Memory, Concurrent Data Structures, Non-blocking, Lock-free

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.50

1 Introduction

Memory is said to be *non-volatile* if it does not lose its contents after a system crash. Non-volatile memory is soon expected to co-exist with or even displace volatile DRAM for main memory (but not caches or registers) in many architectures. As a result, there is increasing interest in the problem of designing and specifying *durable* data structures, that is, data structures whose state can be recovered after a system crash.

A major challenge in designing durable data structures is that caches and registers are expected to remain volatile. Thus, the state of main memory following a crash may be

* This work was supported by the United States - Israel Binational Science Foundation (BSF) grant No. 2012171. Maurice Herlihy was supported by NSF grant 1331141.

inconsistent, missing all previous writes to the data structure that were present in the cache, but not yet written into the main memory. Dealing with arbitrary missing words after a crash requires non-trivial data structure algorithms that make sure key data does get written to main memory (without incurring too much overhead), so that restoration of the data structure to a consistent state becomes possible.

It would be interesting to know if it is possible to build libraries of high performance persistent data structures that are heavily optimized using ad-hoc techniques informed by the data structure architecture and semantics. Previous work focuses on B-tree implementations. The interest in B-trees is natural given their prevalence in file system and database implementations. However, other foundational data structures are also used in application domains that care about persistence; e.g. hash tables in key-value stores, persistent message queues, etc. Since traditional storage media have been block-based, all these applications persist these data structures by marshaling them to a block-based format. Doing so involves non-trivial overhead that was dwarfed by the high cost of disk access. As a result, the in-memory representation and on-disk (-SSD) representation of these data structures are quite different. Byte-addressable persistent memory can be used to create a unified persistent representation. As far as we know, there is no previous work that attempts to optimize these data structures for persistent memory. Furthermore, none of the above works attempt to build highly concurrent, *nonblocking* persistent data structures.

In order to strive for high-performance crash-resilient software on non-volatile memories, we propose to look at modern highly-concurrent data structures, such as the ones used in java.util.concurrent, and enhance them to work with non-volatile memories. Designing such concurrent data structures for upcoming non-volatile memories requires dealing with the challenge of high concurrency and non-volatile durability combined.

We study these challenges by designing a durable version of the lock-free concurrent queue data structure of Michael and Scott [2], which also serves as the base algorithm for the queue in java.util.concurrent. This concurrent data structure is complicated enough to demonstrate the challenges that concurrent durable data structures raise, and simple enough to demonstrate solutions for these challenges.

Recently various definitions were proposed to formalize durability. In this paper we adopt and work with the definition of linearizable durability by Izraelevitz *et al.* [1]. Informally, durable linearizability guarantees that the state of a data structure following a crash reflects a consistent subhistory of the operations that actually occurred. This subhistory includes all operations that completed before the crash, and may or may not include operations in progress when the crash occurred. The main tool for achieving durable linearizability for a concurrent data structure is the use of explicit instructions that force volatile cashed data to be written to non-volatile memory. While such *persistence barrier* instructions enforce correctness, they also carry a performance cost and their use should be minimized.

An alternative, weaker condition, is *buffered durable linearizability*. Informally, this condition guarantees that the state of the object following a crash reflects a consistent subhistory of the operations that actually occurred, but this subhistory *need not* include all operations that completed before the crash.

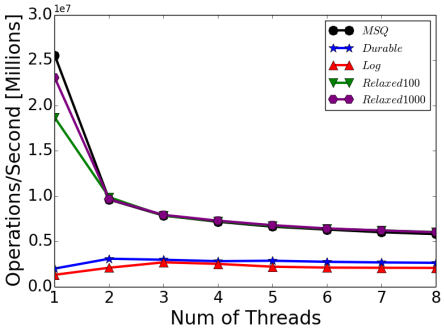
The first main contribution of this paper is the proposal of three novel designs of durable concurrent queues, extending the original Michael-Scott queue for use with non-volatile memory. It is easy to obtain a durable linearizable queue by adding many persistence barrier operations automatically. But, in general, the obtained performance can be very low. In this paper, we attempt to minimize the overhead and still achieve robustness to crashes. The first implementation, denoted *durable* queue, provides durable linearization. The second

implementation, denoted *log* queue, provides durable linearization, as well as an additional property that we discuss next. The third implementation, denoted *relaxed* queue, provides buffered durable linearizability.

When crashes occur during an execution, it is often difficult to tell which operations were executed and which operations failed to execute when a crash occurs. Durable linearizability does not provide a mechanism to determine whether an operation that executed concurrently with a crash was eventually executed. Without the ability to distinguish completed operations from lost operations, it would be difficult to recover the entire program, because in practice it is often important to execute each operation exactly once. In this paper we enable a more robust use of the queue, by defining a new (natural) notion of *detectable execution*. A data structure provides detectable execution if it is possible to tell at the end of a recovery phase whether a specific operation was executed. The *log* queue provides durable linearization and detectable execution. If the program that uses the queue follows a similar procedure for detecting execution, then it is possible to tell how much of the execution has completed on recovery from a crash, and program recovery at higher level becomes possible.

2 Measurements

We have implemented the three queue designs and evaluated their performance by comparing them one against the other and also against the original MS queue. We ran measurements on an 8-cores **Intel** Xeon D-1540 2.6GHz.



Above, we depict the throughput of five queues: *MSQ* is the Michel and Scott's queue, *Durable* is the durable queue, *Log* is the queue that can detect which operations completed before the crash, and *Relaxed* is the queue that only guarantees a view of a prefix of the operations executed before the crash. We ran *Relaxed* with an additional operation *sync* that makes all history durable once every 100 or 1000 operations, denoted *Relaxed100* and *Relaxed1000*. As expected, implementations that provide durable linearization have a noticeable cost, and interestingly, providing detectable execution does not add a significant overhead and may be worthwhile in this case. In addition, implementations that provides only buffered durable linearizability obtain good performance when the `sync()` method is invoked infrequently.

References

1 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium, DISC*, pages 313–327, 2016.

2 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.